Randomized Dynamical Mode Decomposition

Serly Ishkhanian

April 2023

Abstract

The purpose of this project is to study the Randomized Dynamical Mode Decomposition, which uses randomization to improve the efficiency of the DMD algorithm. Randomized methods speed up and make it feasible to perform operations with extremely large datasets that are usually computationally expensive. We will observe the difference between the Singular Value Decomposition (SVD) and randomized methods in how they affect the computational time and the output. To do so, we will perform two experiments: image compression and background-foreground separation of objects in a video.

1 Introduction and Overview

Analyzing and interpreting data are fundamental tasks in many fields, such as Engineering, Social sciences, Finance, Medicine, and many more. However, those come with large datasets, and managing high-dimensional data can be challenging in the sense of computational and time complexity and data storage. In order to facilitate the processing of the data, one uses the assumption that these data have a low-rank structure, implying the existence of interdependency among the columns, to perform dimensionality reduction.

One of the most well-known ways of reducing the dimension of the dataset is the Singular Value Decomposition (SVD) which projects data onto a lower-dimensional space while preserving important information about the data. However, as the dimensions of the dataset increase, processing the data becomes expensive. Thus, computing the SVD becomes inefficient.

The ability to handle high-dimensional datasets efficiently is of great interest, and randomized methods enable one to efficiently perform the same procedures, such as dimensionality reduction and matrix factorization at a fraction of the time it takes for deterministic methods such as the SVD. Not only this, but randomized methods also provide a way of processing large datasets that are too large to be stored in the computer by making it possible to deal with parts of the dataset instead of the whole. Moreover, randomized methods can be utilized to perform the Dynamical Mode Decomposition (DMD) of large datasets efficiently [3]. With applications in fields such as fluid dynamics and engineering, DMD is a data-driven technique that can be applied to high-dimensional data to produce the spatial-temporal modes and a best-fit linear dynamical system that describes how those modes change in time.

In this project, first, we will study the theory of randomized methods and observe how it is used with DMD [3]. We will also perform a few numerical experiments, such as image compression and separation of background and foreground objects in a video, that will allow us to compare the efficincy of randomized methods and the SVD. The theoretical backgrounds of the concepts are in Section 2, and implementation and results of those techniques are in Sections 3 and 4, respectively.

2 Theoretical Background

2.1 Singular Value Decomposition

Given a matrix $D \in \mathbb{R}^{m \times n}$ the singular value decomposition (SVD) of D is:

$$D = U\Sigma V^T, \tag{1}$$

where V^T is the transpose of V, $U \in \mathbb{C}^{m \times m}$, and $V \in \mathbb{C}^{n \times n}$ are unitary matrices (i.e.$U^* = U^{-1}$ and $V^* = V^{-1}$), and $\Sigma \in \mathbb{R}^{m \times n}$ is a rectangular diagonal matrix whose entries are the ordered nonnegative singular values of D such that $\sigma_1 \geq \ldots \sigma_{min\{m,n\}} \geq 0$. Moreover, columns of U are the left singular vectors spanning the column space of D, and columns of V are the right singular vectors spanning the row space of D.

By choosing the first s columns of U and V, and the first s rows and columns of Σ from (1), where $s < min\{m, n\}$, the SVD provides a way of approximating the matrix D with one of lower rank. Thus, the rank-s matrix approximating D is defined as [2], [5]:

$$\hat{D} = U_s \Sigma_s V_s^*. \tag{2}$$

Lastly, using the columns of U_s, we can reduce the dimension of D by projecting D onto the subspace spanned by the columns of U_s. The reduced matrix D_s of size $s \times n$ is computed as:

$$D_s = U_s^* D. \tag{3}$$

2.2 Randomized Methods

Compared to the SVD, randomized methods use randomization to speed up matrix computations and the analysis of large datasets. They also allow the compression of large datasets with a multiplication of a random matrix that preserves the significant structure of the original data.

In [4], the authors state that finding a low-rank approximation of some matrix can be completed in two stages:

- `Stage A:` Let $s \ll min\{m, n\}$ be the target rank and $D \in \mathbb{R}^{m \times n}$ be the data matrix.. Find a matrix $Q \in \mathbb{R}^{n \times s}$ such that $D \approx QQ^T D$.
- `Stage B:` Project the data matrix onto a lower-dimensional space, resulting in $B \in \mathbb{R}^{s \times n}$.

In stage A, the idea is to find a matrix Q of rank$s$, with orthonormal columns, that approximates the column space (the range) of D as accurately as possible. To do so, randomness is used and the process is as follows:

1. Generate s random n-dimensional vector v from the normal Gaussian distribution. This results in the matrix $\Omega \in \mathbb{R}^{n \times s}$ whose columns are the independent s random vectors:
$$\Omega = \begin{bmatrix} v_1 & v_2 & \ldots & v_s \end{bmatrix}. \tag{4}$$
2. By computing $Z = D\Omega$, we obtain the matrix Z whose columns are linearly independent and span the column space of D. Each z_i is a random sample from the range of D. since $z_i = Dv_i$, makes each entry of z_i is a random linear combination of the columns of D.
3. To find Q, compute the QR decomposition of Z, that is $Z = QR$. Columns of Q are orthonormal and its column space approximates the column space of D.

In stage B, using Q, we can compress the high-dimensional data matrix D by projecting it onto the low dimensional space while preserving the structure of the data. $B \in \mathbb{R}^{s \times n}$ is defined as

$$B = Q^T D \in \mathbb{R}^{s \times n}. \tag{5}$$

From this, we observe that

$$D \approx QB. \tag{6}$$

This process will be referred to as the QB decomposition.

2.2.1 Oversampling

In stage A, when choosing the target rank s, D may not have that as an exact rank, and if it does not, then we can have $\{\sigma_i \neq 0\}_{i \geq s+1}$. Thus, it is not effective to generate $\Omega \in \mathbb{R}^{s \times n}$. So, to capture the column space of D and construct a better basis, oversample by generating $l = s + p$ random vectors instead of s, where $p = \{5, 10\}$ commonly. This provides a method of improving the approximations obtained by the QB decomposition.

2.2.2 Power Iterations

Since it is possible that D may have singular values decaying slowly, a method of power iterations can be performed to speed up the decaying rate and also improving the approximations. Hence, D is preprocessed by

$$D^{(q)} = (DD^T)^q D, \tag{7}$$

where q is an integer defining the number of iterations. In practice, commonly,$q = \{1, 2\}$. Lastly, the sampling matrix Z is computed as

$$Z = ((DD^T)^q D)\Omega. \tag{8}$$

2.3 Blocked Randomized Methods

Storage issues can arise when applications contain extremely large datasets, such as millions of rows and hundreds and thousands of columns. Randomized methods provide a way of dealing with the data by focusing on parts of it at a time. Moreover, it is also possible to deal with the data and perform computations on different computers, then combine them all together.

In [3], the authors proceed by dividing the data matrix D into b blocks $D_i \in \mathbb{R}^{\frac{m}{b} \times n}$along the rows. Then, each block D_i is approximated by a rank-s matrix using the QB decomposition (in practice, the target rank is chosen as $l = s+p$). For instance, assume $b = 3$. D is partitioned as follows

$$D = \begin{bmatrix} D_1 \\ D_2 \\ D_3 \end{bmatrix} \approx \begin{bmatrix} Q_1 B_1 \\ Q_2 B_2 \\ Q_3 B_3 \end{bmatrix} = \text{diag}(Q_1, Q_2, Q_3) \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix}. \tag{9}$$

Combining all $B_i \in \mathbb{R}^{s \times n}$, we obtain

$$K = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} \in \mathbb{R}^{b \cdot s \times n}. \tag{10}$$

The matrix K has reduced dimensions, however, by applying the QB decomposition of K

$$K \approx \hat{Q} B, \tag{11}$$

the projected matrix $B \in \mathbb{R}^{s \times n}$ of desired dimension is obtained. Lastly, the basis matrix $Q \in \mathbb{R}^{m \times s}$ is computed as

$$Q = \text{diag}(Q_1, Q_2, Q_3)\hat{Q}. \tag{12}$$

2.4 Randomized Dynamical Mode Decomposition

Now that we have discussed randomized methods, we will observe how randomization is utilized in [3] to efficiently perform the DMD. First, let's recall what DMD is.

2.4.1 Dynamical Mode Decomposition

Given a data matrix D whose columns represent a sequence of snapshots $x_0, x_1, \ldots, x_n \in \mathbb{R}^m$ of the state of a system as it is changing in time, the goal of the DMD is to extract spatial-temporal patterns that describe the behavior of the system over time. This is done by first separating the data matrix into X and $Y \in \mathbb{R}^{m \times n}$ as follows:

$$X = [x_0 \quad x_1 \quad \ldots \quad x_{n-1}], \quad Y = [x_1 \quad x_2 \quad \ldots \quad x_n], \tag{13}$$

where columns of X represent all the snapshots from the first to before the last. Similarly, columns of Y represent the snapshots that are shifted one time frame into the future. Moving from one snapshot to another could be a nonlinear mapping. However, DMD attempts to find the best linear map, i.e a matrix $A \in \mathbb{R}^{s \times s}$, that allows one to move from X to Y such that we can express Y as follows [1]:

$$Y \approx AX. \tag{14}$$

Finding A leads to solving a minimization problem [1]:

$$\underset{A \in \mathbb{R}^{s \times s}}{\operatorname{argmin}} \quad \|Y - AX\|_F^2, \tag{15}$$

where $\|\cdot\|_F$ represents the Frobenius norm of a matrix; which is the square root of the sum of the squares of the elements of a matrix. It is of interest to find a matrix A so that entries of AX are very close to Y, and it turns out that the exact solution of (15) is:

$$A = YX^\dagger, \tag{16}$$

where $X^\dagger$ is the pseudoinverse of X.

2.4.2 Randomized DMD

The issue arises when attempting to compute the DMD matrix A. Hence, the authors in [3] introduce the concept of randomized DMD that reduces the dimension of a large data matrix D and performs DMD effectively.

Using the randomized methods, the basis matrix $Q \in \mathbb{R}^{m \times l}$ approximating the column space of D is generated, and D is projected onto a low dimensional space by

$$B = Q^T D = \begin{bmatrix} b_0 & b_1 & \ldots & b_n \end{bmatrix}, \tag{17}$$

where each column $b_i \in \mathbb{R}^l$ represents the low-dimensional snapshot. Now, using the low dimensional matrix, the matrices X and $Y \in \mathbb{R}^{l \times n}$ are constructed:

$$X = \begin{bmatrix} b_0 & b_1 & \ldots & b_{n-1} \end{bmatrix}, \quad Y = \begin{bmatrix} b_1 & b_2 & \ldots & b_n \end{bmatrix}. \tag{18}$$

Following the process in Section 2.4.1, the linear map $A \in \mathbb{R}^{l \times l}$ is estimated as

$$\hat{A_B} = YX^\dagger. \tag{19}$$

Then $\hat{A_B}$ is projected onto a subspace spanned by the columns of U_x, as follows:

$$A_B = U_x^T \hat{A_B} U_x, \tag{20}$$

where the SVD of $X = U_x \Sigma_x V_x^T$. $U_x \in \mathbb{R}^{l \times s}$, $V_x \in \mathbb{R}^{s \times n}$, and $\Sigma \in \mathbb{R}^{s \times s}$ contains the singular values. By computing the eigendecomposition of A_B, we obtain the DMD modes that contain information regarding the underlying system:

$$A_B W_B = W_B \Lambda, \tag{21}$$

where the columns of $W_B \in \mathbb{C}^{l \times l}$ are the eigenvectors $\phi \in \mathbb{C}^l$ are the spatial modes of the data representing the most dominant patterns in the data, and $\Lambda \in \mathbb{R}^l$ is a diagonal matrix whose entries represent the eigenvalues μ which are discrete time dynamics that inform us how the modes change in time.

2.5 Background and Foreground Separation

Once we obtain the eigendecomposition of A_B, we obtain essential information about the data. One of the tasks we can perform is background and foreground of objects in videos by separating the fast and slow modes of the DMD matrix.

Let $\omega_i = \ln(\mu_i)/dt$ represent the continuous time eigenvalues, where dt = video duration/number of frames. The modes associated with the eigenvalues $\mid \omega_k \mid \approx 0$ are the slow modes (the background) since they don't change much in time and the remaining modes $\mid \omega_j \mid$ that are bounded away from zero for all $j \neq k$, where $k \in \{1, 2, \ldots, l\}$ represent the fast modes (the foreground objects) [1]. In Section 3, we will set a threshold $\lambda > 0$ to distinguish between the modes. DMD reconstruction of B is $B_{DMD} \in \mathbb{R}^{l \times N}$, where $N = n + 1$, whose columns are the reconstructed frames [1]:

$$B_{DMD}(t) = c_p \phi_p e^{\omega_p t} + \sum_{j \neq p} c_j \phi_j e^{\omega_j t}, \tag{22}$$

where each c_i is the initial amplitude of each mode. So, the vector $c \in \mathbb{C}^l$ can be found by $c = W_B^{\dagger} b_0$ [1], since we know that at $t = 0$, (22) is b_0. Although each term of (22) is complex, they add up to a real-valued matrix. Thus, when dividing the DMD terms into its low-rank and sparse reconstructions, it is essential to be careful with the complex elements as it affects the accuracy. Consider the reconstruction of the approximate low-rank that corresponds to the background video [1]:

$$B_{Low-Rank} = c_k \phi_k e^{\omega_k t}, \tag{23}$$

and from the fact that $B = B_{Low-Rank} + B_{Sparse}$, we can obtain the DMD's approximate sparse approximation:

$$B_{Sparse} = \sum_{j \neq k} c_j \phi_j e^{\omega_j t}, \tag{24}$$

which can be calculated with real-valued elements:

$$B_{Sparse} = B - \mid B_{Low-Rank} \mid, \tag{25}$$

where $\mid \cdot \mid$ denotes the modulus of each element in the matrix. One issue that could arise is that (25) could produce negative value pixel intensities. To counteract this, a matrix $R \in \mathbb{R}^{l \times N}$ is created to contain those negative values and make the following adjustments to account for the magnitudes of the complex values from the DMD reconstruction:

$$B_{Low-Rank} \mid \leftarrow R + \mid B_{Low-Rank} \mid, \quad \text{and} \quad B_{Sparse} \leftarrow B_{Sparse} - R. \tag{26}$$

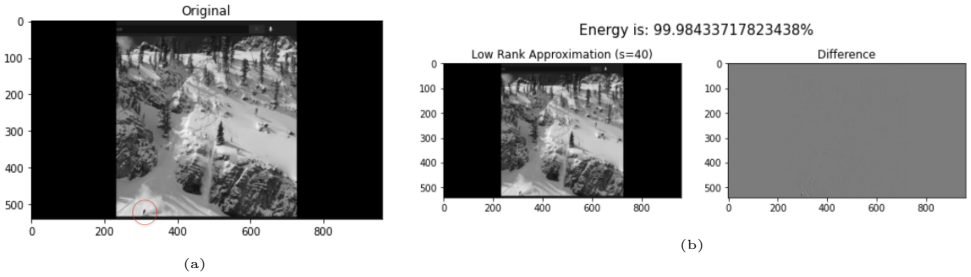
This ensures that the approximate low-rank and sparse DMD matrices are real-valued [1].

3 Algorithm Implementation and Development

3.1 Setup

Before implementing the above techniques, we must load the Ski Drop video. It is a 4D dataset, and we transform the it into a 3D datasets by converting it to greyscale. The video contains 454 frames of size 540 by 960. Then, we turn the 3D dataset into a 2D matrix by vectorizing the frames of the video, so that as we go across the columns of the data matrix D, it will be as if we are moving through time frame by frame. Thus, we obtain the data matrix $D \in \mathbb{R}^{518400 \times 454}$.

Since this is a high-dimensional dataset, performing certain operations using D directly can be computationally expensive. Hence, we will first reduce the dimension of the data matrix D using the SVD and randomized methods. Then, we will obtain the DMD matrix and analyze its eigendecomposition.



(a) (b)

Figure 1: Image in (a) is the image of the original frame 425 from the video (425^{th} frame is used since the individual in the video is visible at the bottom). (b) shows the result of the low-rank approximation of the 425^{th} frame of the Ski Drop video using $s = 30$. The right column illustrates the difference between the original frame and low-rank approximation.

3.2 Dimensionality Reduction: SVD

First, we compute the truncated SVD of D to produce a low-rank approximation of D. To choose the value of s, we observe the cumulative energy of the first s singular values by computing:

$$\text{cumulative energy} = \frac{\sum_{i=1}^{s} \sigma_i^2}{\sum_{i=1}^{min(m,n)} \sigma_i}. \tag{27}$$

This is a measure that tells us how much of the data can be explained by the s singular values. In Appendix C, results of the low-rank approximations using different values of s can be found. However, observe from Figure 1 that using only $s = 30$ with 99.979% energy captured for the Ski video, we can obtain a satisfactory low-rank approximation. Using those values of s, we obtain the projected dataset $D_s = U_s^T D \in \mathbb{R}^{s \times n}$. In Section 4, we will observe the speed of this process.

3.3 Dimensionality Reduction: Randomized Methods

Using the same values of s we also reduce the dimension of D by using the randomized methods mentioned in Section 2 and following Algorithm 1. We will also use Algorithm 1 to project each block of the dataset when using the blocked randomization method. For more details on partitioning the data, please view the code in Section B.

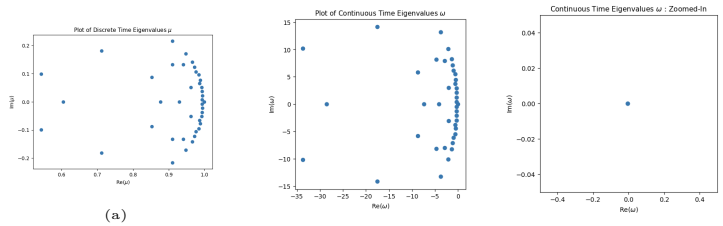
Algorithm 1: Randomized QB Decomposition

Set $l = s + p$
Generate a random matrix $\Omega = \text{np.random.normal}(0, 1, \text{size} = (n, l))$
Compute the sampling matrix $Z = D\Omega$
Perform power iterations (optional)
for $j = 1, \ldots, q$ **do**
  $Z = D(D^T Z)$
end for
Obtain the QR decomposition $Z = QR$
Compute the projected data matrix $B = Q^T D$

3.4 DMD

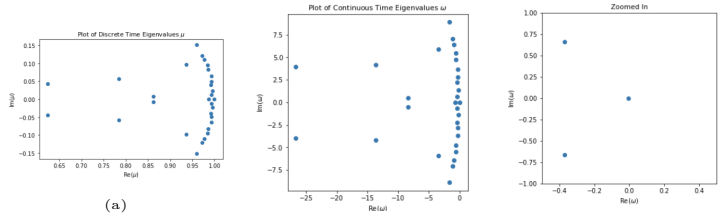
Once we obtain the data matrices D_s and B that are projected onto their respective lower-dimensional spaces, from the three different methods, we will apply the DMD technique to extract the spatial-temporal patterns of the data.

Following the theory in Section 2, we compute the matrices X and Y of the projected data matrices. Then, we compute the DMD matrix A for each method accordingly.



(a) (b) (c)

Figure 2: Plots of the discrete and continuous time eigenvalues of DMD matrix obtained from the QB decomposition (with $s = 30$, $p = 10$, and $q = 2$) of the Ski Drop video. (c) shows the zoomed in version of (b) near 0.



(a) (b) (c)

Figure 3: Plots of the discrete and continuous time eigenvalues of DMD Matrix of Ski Drop video. (c) shows the zoomed in version of (b) near 0.

3.4.1 Background-Foreground Separation

After obtaining A, we compute its complex eigenvalues and eigenvectors.

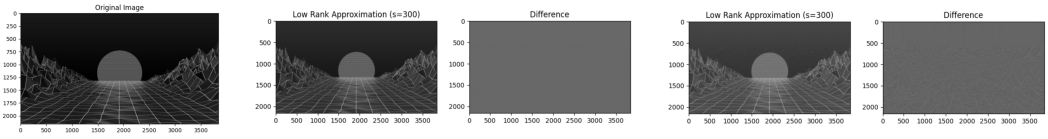
To distinguish between the slow modes and fast modes, we set the value of the threshold λ by observing Figures 2 and 3 of the continuous time eigenvalues near 0 (as mentioned in Section 2.5. We can set $\lambda = 0.01$ as the cut off point since the points in both figures 2c and 3c are above that value. Using this threshold, we can distinguish between the slow modes and fast modes, and create our low-rank background and sparse foreground following Algorithm 2. For more steps, please view the detailed algorithm in Appendix B.

Algorithm 2: DMD Process After Obtaining DMD Matrix A

Get eigenvalues and eigenvectors of A
Compute $c = \Phi^{\dagger} b_0$
Compute w_i for $i = 1, \ldots, s$
for $i = 1 : s$ **do**
  Get indices i such that $\mid \omega_i \mid < 0.01$
  Create a vector c_{slow} whose entries are all zeros except at the positions of those indices
  Set $c_{slow}[i] = c[i]$
end for
Create $B_{LowRank}$
Create $B_{Sparse} = B - B_{LowRank}$
Create matrix R whose entries are all zeros except the negative values of B_{Sparse}
Reassign $B_{Sparse} \leftarrow B_{Sparse} - R$
Reassign $B_{LowRank} \leftarrow \mid B_{LowRank} \mid + R$
Plot images of $B_{LowRank}$ and B_{Sparse}

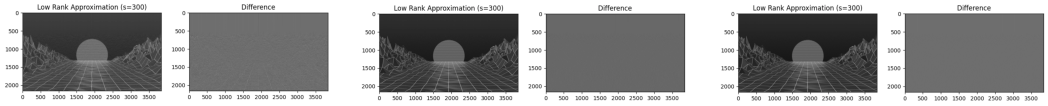
4 Computational Results

To observe the effect of the randomized methods, two numerical experiments were conducted. The first experiment was applied on a small-dimensional image of size 2160×3840. It was of interest to see the speed difference between the randomized QB decomposition, and the SVD.



(a) (b) The SVD (c) Case: $s = 30$, $p = 0$, $q = 0$.

Figure 4: Figure (a) is the original image that will be used to apply and compare the randomized methods and the SVD, (b) represents the low-rank approximation obtained from the SVD using $s = 300$, and (c) represents the low-rank approximation obtained from the QB decomposition method using $s = 30$ without oversampling and power iteration. The right columns in (a) and (b) show the difference between the original image and the low-rank approximation.



(a) Case: $s = 30$, $p = 10$, $q = 0$. (b) Case: $s = 30$, $p = 10$, $q = 1$. (c) Case: $s = 30$, $p = 10$, $q = 2$.

Figure 5: (a), (b), and (c) represent the low-rank approximation obtained from the QB decomposition method using $s = 30$ and $p = 10$ as we change q. The right columns in (a) and (b) show the difference between the original image and the low-rank approximation.

4.1 Experiment 1: Image Compression

At first, when studying the randomized methods, I applied the techniques on a small data matrix and observed their effects. Thus, experiment 1 tests the result and speed of the randomized QB decomposition method, the block randomized method, and the SVD in the application of image compression.

Using Figure 4a as the original image, it corresponds to the data matrix $D \in \mathbb{R}^{2160\times3840}$. Figure 4 shows the low-rank approximation of D which is of size 300×3840 computed by applying the SVD and the QB decomposition without oversampling and power iteration. Although both Figures 4b and 4c provide good low-rank approximations of the original data matrix D, there is a slight difference in the intensity of the colors in the image. We proceed by altering the oversampling parameter p and the power iteration parameter q. Figure 4 shows the results of those changes. Figure 5 illustrates the effect of increasing the parameter q. We observe that the approximations, in terms of the intensity of the colors, are improving. The time it took to perform those operations for each method are:

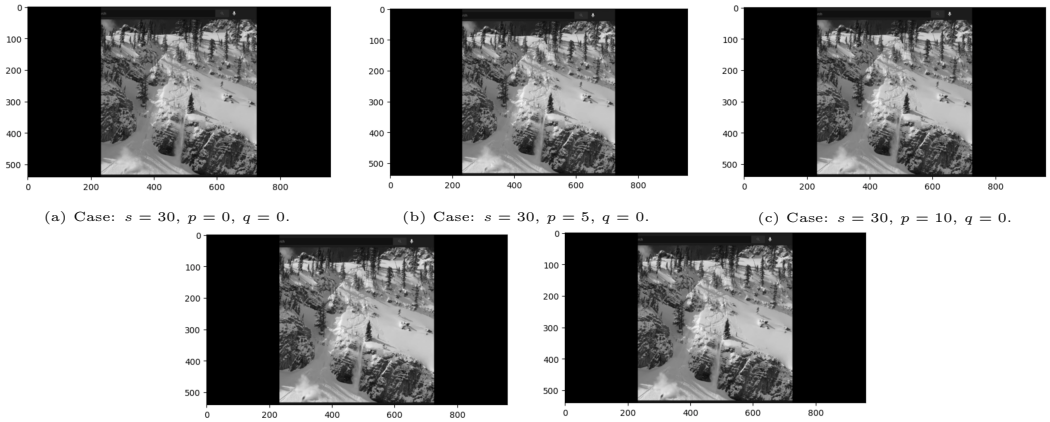
1. The SVD with $s = 300$, was completed in approximately 34 seconds.
2. Case 1 with $s = 300$, $p = 0$, and $q = 0$, was completed in approximately 0.55 seconds.
3. Case 3 with $s = 300$, $p = 10$, and $q = 0$, was completed in approximately 0.62 seconds.
4. Case 4 with $s = 300$, $p = 10$, and $q = 1$, was completed in approximately 1 second.
5. Case 5 with $s = 300$, $p = 10$, and $q = 2$, was completed in approximately 1.5 seconds.
6. The blocked randomized method with $s = 300$, $p = 10$, and $q = 1$, was completed in approximately 2 seconds. Figure 10 in Appendix C, shows the results.

Clearly, the randomized methods are faster than the SVD when reducing the dimensions.

4.2 Experiment 2: Background-Foreground Separation

Using the Ski Drop video dataset, we also observe that randomized methods are faster in constructing the projected datasets B compared to the truncated SVD.

Using the randomized method, the QB decomposition from Algorithm 1, we conduct five cases where we observe the performance and computation time of the procedure as we change the values of the oversampling parameter p and the power iteration parameter q while keeping $s = 30$.



(a) Case: $s = 30$, $p = 0$, $q = 0$.

(b) Case: $s = 30$, $p = 5$, $q = 0$.

(c) Case: $s = 30$, $p = 10$, $q = 0$.

(d) Case: $s = 30$, $p = 10$, $q = 1$.

(e) Case: $s = 30$, $p = 10$, $q = 2$.

Figure 6: Using a target rank $s = 30$, (a), (b), and (c) illustrate the low-rank approximation of the data matrix D as we change the oversampling parameter p. Figures (d) and (e) represent the reconstruction of the low-rank approximation of D with $p = 10$ as we change the power iteration parameter q.

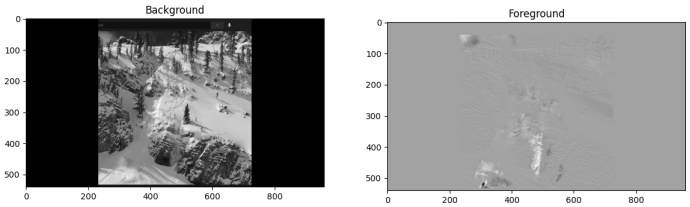
Figure 6 illustrates the low-rank approximation of the data matrix D using a target rank $s = 30$. Compared to the SVD, Figure 6a shows that the low-rank approximation matrix QB without oversampling and power iterations is not good since the individual in the video is missing. Notice how as the oversampling parameter increase in Figures 6b and 6c, the approximations also improve. However, both figures contain blurriness, and the individual is still not fully visible. On the other hand, observe how when power iterations are performed with oversampling, the approximations in Figures 6d and 6e become better. Now, the individual is clearly visible. These results fit with the statement that oversampling and power iterations improve the performance of the randomized method.

In terms of the computation times for the randomized methods, we obtain that:

1. Case 1 with $s = 30$, $p = 0$, $q = 0$, was completed in approximately 5.27 seconds.
2. Case 2 with $s = 30$, $p = 5$, $q = 0$, was completed in approximately 5.75 seconds.
3. Case 3 with $s = 30$, $p = 10$, $q = 0$, was completed in approximately 6.44 seconds.
4. Case 4 with $s = 30$, $p = 10$, $q = 1$, was completed in approximately 9.18 seconds.
5. Case 5 with $s = 30$, $p = 10$, $q = 2$, was completed in approximately 11.76 seconds.

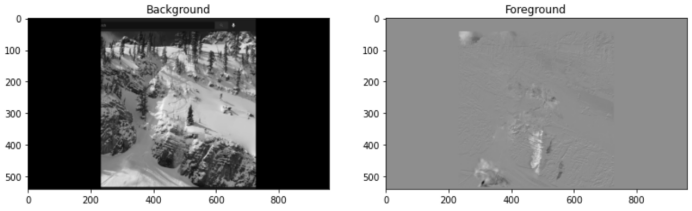
Whereas the low-rank approximation obtained from the SVD, required approximately 61.68 seconds. The difference in computation time is very apparent. Additionally, the blocked randomized method was also utilized to perform dimensionality reduction with $b = 4$, $s = 30$, $p = 10$, and $q = 2$. The process took approximately 13 seconds. If the number of power iterations is reduced, then it will require a shorter period. The only issue faced with the blocked randomized method in this application was that Python crashed when attempting to compute the matrix Q.

Figures 7 and 8 show the results of applying DMD for background and foreground object separation done on frame 425 of the Ski Drop video using the randomized method and the SVD, respectively. As we can see, in Figure 7, the algorithm was able to separate the ski man and some falling snow (foreground objects) from the background cleanly. Although the separation in Figure 8 is also very good, a slight shadow in the background figure is slightly visible if looked closely. Overall, the algorithm used for the randomized methods outperformed in separating the background and foreground objects in the Ski Drop video.



(a) (b)

Figure 7: Figures (a) and (b) represent the result of background-foreground separation in the Ski Drop video using the projected data matrix obtained from the QB decomposition.



(a) (b)

Figure 8: Figures (a) and (b) represent the result of background-foreground separation in the Ski Drop video using the projected data matrix obtained from the SVD.

5 Summary and Conclusions

In this project, we observed how the authors [3] developed Randomized Dynamical Mode Decomposition by using randomized methods to improve the efficiency of the known algorithms. We saw that randomized methods produce a low-rank approximation of a large data matrix rapidly and efficiently with a multiplication of a random matrix. Moreover, by partitioning a large dataset, randomized methods also provide a solution to overcoming storage solutions. Lastly, from the two experiments conducted, we observed how randomized methods provide the low-rank approximation matrices at a fraction of the time it takes using the SVD method. We also saw how randomized DMD outperforms in the background-foreground separation of a video compared to the performance of the DMD with the SVD used.

References

[1] Jason J. Bramburger. *Data-Driven Methods for Dynamical Systems.* Concordia University, 2023.

[2] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[3] N Benjamin Erichson, Lionel Mathelin, J Nathan Kutz, and Steven L Brunton. Randomized dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 18(4):1867–1891, 2019.

[4] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[5] Leon Mirsky. Symmetric gauge functions and unitarily invariant norms. *The quarterly journal of mathematics*, 11(1):50–59, 1960.

Appendix A Python Functions

- `skvideo.io.vread(video.mp4)` returns a 4D array of the video.

- `skvideo.utils.rgb2gray(videodata)` computes the grayscale video from the input video returning the standardized shape (number of frames, height, width, 1).
- `reshape` function will reshape an array into the dimension you want. We use it to vectorize the frames.
- `u,s,vh = np.linalg.svd(M,full_matrices = False)` produces the compact SVD of the matrix M.
- `plt.imshow` takes a 2D array (in our case, a frame) and produces a picture of the frames.
- `sum` will add the numbers in an array.
- `np.delete(array,n,axis=1)` deletes the n-th column of array.
- `p.linalg.pinv(X)` computes the pseudoinverse of matrix X.
- `np.linalg.eig(A)` returns the eigenvalues and eigenvectors of matrix A.
- `np.abs(element)` will take the modulus of a complex element.
- `cv2.imread('image path')` Loads image.
- `cv2.cvtColor()` Used to convert an image from one color space to another
- `np.random.normal()` Draws random samples from a normal Gaussian distribution.
- `np.block()` Takes in a list of arrays and combines them according to a specified block structure.
- `time.time()` Returns the time as a floating point number expressed in seconds. Used for computing computational time.

Appendix B Python Code

```
import numpy as np
import cv2
import time
from matplotlib import pyplot as plt
from matplotlib import pyplot as plt
from PIL import Image

pip install scikit-video

import skvideo.io

"""#The QB decompsoition function """

def randomized(m, s,q,p):

  l = s + p
  n = m.shape[1]

  omega = np.random.normal(0, 1, size=(n, l))

  Z = m @ omega

  for k in range(q):
```

```
    Z = m @(m.T @Z)

  Q,R = np.linalg.qr(Z)
  B = Q.T @ m

  return Q,B

"""#SVD- Data projection"""

def low_rank_approximation(number_of_singularvalues):

  # Compute the projected data
  new_matrix = u[:,0:number_of_singularvalues].conj().T @ Transpose_frames

  return new_matrix

"""#Experiment 1"""

# Upload image
image = cv2.imread('/content/JFeEHc.jpg')

# Read image
image_data = image.copy()

# Get image size
image.shape

# Convert to grayscale
gray_image = cv2.cvtColor(image_data, cv2.COLOR_BGR2GRAY)

#plot the approximated image and the difference bwteen approximated and original

def plot(f1, new_matrix, s) :

  reduced_frame = np.reshape(f1,(2160,3840))

  fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(10,8))
  ax1.imshow(reduced_frame,cmap='gray')
  ax1.set_title("Low Rank Approximation (s="+str(s)+")")
  ax2.imshow(np.reshape(gray_image-f1,(2160, 3840)),cmap='gray')
  ax2.set_title(" Difference ")


  plt.show()

plt.imshow(gray_image,cmap='gray')
plt.title("Original Image")

"""The SVD"""

start = time.time()
f1, new_matrix = low_rank_image_reconstruction(300)
end = time.time()
```

```
# Compute time it took
final = end-start


# Plot the reconstructed image
plot(f1,new_matrix,300)

print("\nTime it took to compute the low rank approximation: ", final,"seconds")

"""Randomized method """



"""Case: s=300, q=0, p=0"""

start_time = time.time()

Q1,B = randomized(gray_image,300,0,0)

end_time = time.time()
final_time = end_time - start_time
print(final_time)

plot(Q1@B,Q1.T@gray_image,300)

"""Case: s=300, q=0, p=10"""

start_time = time.time()

Q1,B = randomized(gray_image,300, 0,10)

end_time = time.time()
final_time = end_time - start_time
print(final_time)

plot(Q1@B,Q1.T@gray_image,300)

"""Case: s=300, q=1, p=10"""

start_time = time.time()

Q1,B = randomized(gray_image,300,1,10)

end_time = time.time()
final_time = end_time - start_time
print(final_time)

plot(Q1@B,Q1.T@gray_image,300)

"""  Case: s=300, q=2, p=10"""

start_time = time.time()

Q1,B = randomized(gray_image,300,2,10)
```

```
end_time = time.time()
final_time = end_time - start_time
print(final_time)

plot(Q1@B,Q1.T@gray_image,300)

"""# Block Randomized Algorithm

"""

b  = int(2160/4)
s=300
q=1
p=10
D1 = gray_image[:b,:]
D2 = gray_image[b:2*b,:]
D3 = gray_image[2*b:3*b,:]
D4 = gray_image[3*b:4*b,:]

# Approximate each block
rstart_time1 = time.time()
Q1,B1 = randomized(D1,s,q,p)
Q2,B2 = randomized(D2,s,q,p)
Q3,B3 = randomized(D3,s,q,p)
Q4,B4 = randomized(D4,s,q,p)


K = np.vstack((B1,B2,B3,B4))
Qhat,B = randomized(K,s,q,p)

rend_time1 = time.time()

rfinal_time1 = rend_time1 - rstart_time1
print(rfinal_time1)

# Create the block diagonal matrix
n = gray_image.shape[1]
I = np.zeros((540,s+p))

Q = np.block([[Q1,I,I,I],[I,Q2,I,I],[I,I,Q3,I],[I,I,I,Q4]])@Qhat
Q.shape

plot(Q@B,Q.T@gray_image,300,10)

"""# Experiment 2"""

videodata = skvideo.io.vread("/content/ski_drop_low.mp4")

# Get dimensions
number_of_frames, height, width, colors = videodata.shape

# Convert to greyscale
grey_ski = skvideo.utils.rgb2gray(videodata)

# Create a 2D data matrix
```

```
frames = []

for frame in grey_ski:

  #Vectorize the frames by stacking the columns vertically
  reshaped_frame = frame.reshape((540*960))

  frames.append(reshaped_frame)        #adds them to the matrix "frames" as rows



# Transpose the matrix frames so that the each column represents a frame and dimension becomes sp
Transpose_frames = np.array(frames).T

start = time.time()
u,s,vh = np.linalg.svd(Transpose_frames,full_matrices = False)
end = time.time()

final= end-start
print("Time it took : ", final ," seconds.")

# Projected Matrix for the SVD
start1 = time.time()

D30 = low_rank_approximation(30)
end1 = time.time()
final1 = end1-start1

print("Time it took : ", final1 ," seconds.")

total = final +final1

print("Total time to project data onto a lower-dimensional space using the SVD is:",total, "secon

"""Case s=30,p=5,q=0 """

rstart_time = time.time()

Q0,B0 = randomized(Transpose_frames,30,0,5)
rend_time = time.time()

rfinal_time = rend_time - rstart_time
print(rfinal_time)

# the low-rank approximation of D
approx= Q0@B0

# Accuracy of the approximation
plt.imshow(np.reshape(approx[:,425],(height,width)),cmap="gray")

"""Case s=30, q=0, p=10"""

rstart_time = time.time()

Q0,B0 = randomized(Transpose_frames,30,0,10)
```

```
rend_time = time.time()

rfinal_time = rend_time - rstart_time
print(rfinal_time)

# the low-rank approximation of D
approx= Q0@B0

# Accuracy of the approximation
plt.imshow(np.reshape(approx[:,425],(height,width)),cmap="gray")

"""Case s=30, q=1, p=10"""

rstart_time = time.time()

Q0,B0 = randomized(Transpose_frames,30,1,10)
rend_time = time.time()

rfinal_time = rend_time - rstart_time
print(rfinal_time)

# the low-rank approximation of D
approx= Q0@B0


# Accuracy of the approximation
plt.imshow(np.reshape(approx[:,425],(height,width)),cmap="gray")

"""Case : s=30, p=10, q=2"""

rstart_time = time.time()

Q0, B0 = randomized(Transpose_frames,30,2,10)
rend_time = time.time()

rfinal_time = rend_time - rstart_time
print(rfinal_time)

# the low-rank approximation of D
approx= Q0@B0


# Accuracy of the approximation
plt.imshow(np.reshape(approx[:,425],(height,width)),cmap="gray")

plt.imshow(np.reshape((u[:,:s] @D30)[:,425],(height,width) ),cmap="gray")

"""#Blocked randomized method"""

# Partition into 4 blocks along the rows
b  = int(518400/4)

D1 = Transpose_frames[:b,:]
D2 = Transpose_frames[b:2*b,:]
D3 = Transpose_frames[2*b:3*b,:]
```

```
D4 = Transpose_frames[3*b:4*b,:]

# Approximate each block using the qb decomposition with s=30 , p = 10, q=2
s=30
p=10
q=2
rstart_time1 = time.time()
Q1,B1 = randomized(D1,s,q,p)
Q2,B2= randomized(D2,s,q,p)
Q3,B3 = randomized(D3,s,q,p)
Q4,B4 = randomized(D4,s,q,p)

K = np.vstack((B1,B2,B3,B4))
Qhat, B_block = randomized(K,s,q,p)
rend_time1 = time.time()

rfinal_time1 = rend_time1 - rstart_time1
print(rfinal_time1)

# Create the block diagonal matrix
n = Transpose_frames.shape[1]
I = np.zeros((129600,b*(s+q)))

Q = np.block([[Q1,I,I,I],[I,Q2,I,I],[I,I,Q3,I],[I,I,I,Q4]])@Qhat
Q.shape

# Returns the X, Y matrices and the DMD matrix A
def create_XY(data):
  n = data.shape[1]
  X = np.delete(B0,n-1,axis=1)
  Y = np.delete(B0,0,axis=1)

  A = Y @ np.linalg.pinv(X)

  return X, Y, A

# Create the matrices X and Y for randomized method
X,Y,A_b = create_XY(B0)

# For randomized methods
# Compute the SVD of X
u,s,vh = np.linalg.svd(X)
A = u.T @A_b @u

# With SVD directly compute
# Create the matrices X and Y and the DMD Matrix A
X,Y,A = create_XY(D30)

"""# Dimensionality Reduction: Randomized Methods

Case s= 30, q=0, p=0
"""

rstart_time = time.time()
```

```
Q0,B0 = randomized(Transpose_frames,30,0,0)

rend_time = time.time()
rfinal_time = rend_time - rstart_time
print(rfinal_time)

# the low-rank approximation of D
approx = Q0@B0

# Accuracy of the approximation
plt.imshow(np.reshape(approx[:,425],(height,width)),cmap="gray")

#Define
s = 30
p = 10
l = s+p

# Get eigenvalues and eigenvectors of A
evalues, evectors = np.linalg.eig(A)

# Create omega: continuous time eigenvalues of A
omega =[]

# duration of video/number of frames
dt =8/number_of_frames

for eigenvalue in evalues:
  omegak = np.log(eigenvalue)/dt
  omega.append(omegak)

omega = np.array(omega)



"""Plot the discrete time eigenvalues """

plt.scatter(evalues.real,evalues.imag)
plt.xlabel("Re($\mu$)")
plt.ylabel("Im($\mu$)")
plt.title("Plot of Discrete Time Eigenvalues $\mu$",fontsize=11)
plt.show()


"""Plot the continuous time eigenvalues

(This will be used to determine a threshold to separate the fast and slow ones) """

plt.figure(figsize=(5,5))
plt.xlabel("Re($\omega$)")
plt.ylabel("Im($\omega$)")
plt.title("Plot of Continuous Time Eigenvalues $\omega$",fontsize=11)
plt.scatter((omega).real,(omega).imag)
plt.show()

# Zoomed in
```

```
plt.figure(figsize=(5,5))
plt.xlabel("Re($\omega$)")
plt.ylabel("Im($\omega$)")
plt.title("Continuous Time Eigenvalues $\omega$ : Zoomed-In",fontsize=11)
plt.axis([-0.5,0.5, -0.05, 0.05])
plt.scatter((omega).real,(omega).imag)
plt.show()


# Find coefficients of the eigendecomposition
b = np.linalg.pinv(evectors) @ X[:,0]
b.shape


"""Find index of the continuous time eigenvalues that fall below the threshold"""

slow_index = []
for i in range(len(omega)):
  if abs(omega[i])<0.01:
    slow_index.append(i)



"""Create a vector b corresponding to the coefficients of the slow modes that has zeros everywher

b_slow = np.zeros(b.shape).astype(complex)

for i in slow_index:
  b_slow[i] = b[i]

np.array(slow_index).shape



"""The background (slow_x)"""

slow_x = np.zeros((l,number_of_frames)).astype('complex')

for i in range(number_of_frames):
  for j in range(l):
    if b_slow[j]!=0:
      slow_x[:,i] = slow_x[:,i]+(b[j]* np.exp(omega[j]*i))*evectors[:,j]



"""The foreground (x_sparse)"""

x_sparse = B0 - np.abs(slow_x)



"""Calculate the matrix R """
#Since the above calculation may result in x_sparse having negative values in some of its element
```

```
new_r =np.zeros(x_sparse.shape)

for i in range(number_of_frames):
  for j in range(l):
    if x_sparse[j,i].real <0:
      new_r[j,i] = x_sparse[j,i]


# Reassign
new_sparse = x_sparse - new_r

new_slow =np.abs(slow_x) + new_r


#Plot the foreground and background
plt.imshow(np.reshape((Q0@ new_sparse[:,425]),(height, width)),cmap = 'gray')
plt.title("Foreground")
plt.show()

plt.imshow(np.reshape((Q0@ new_slow[:,425]),(height, width)),cmap = 'gray')
plt.title("Background")
plt.show()

# Note: the code "ski_video" can be used to produce the cumulative enrgies and plots of the low-r
```

Appendix C Additional Plots

- Figure 9 demonstrates the low-rank approximation of the Ski Drop dataset using different number of singular values.
- Figure 10 demonstrates the result of the blocked randomized method with $b = 4$, $s = 30$, $p = 10$, $q = 1$.

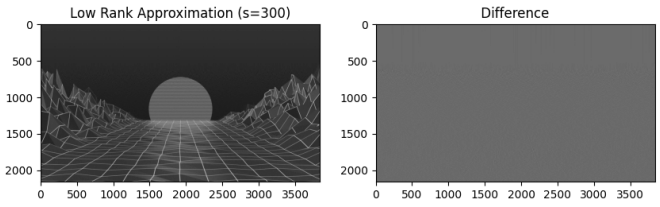
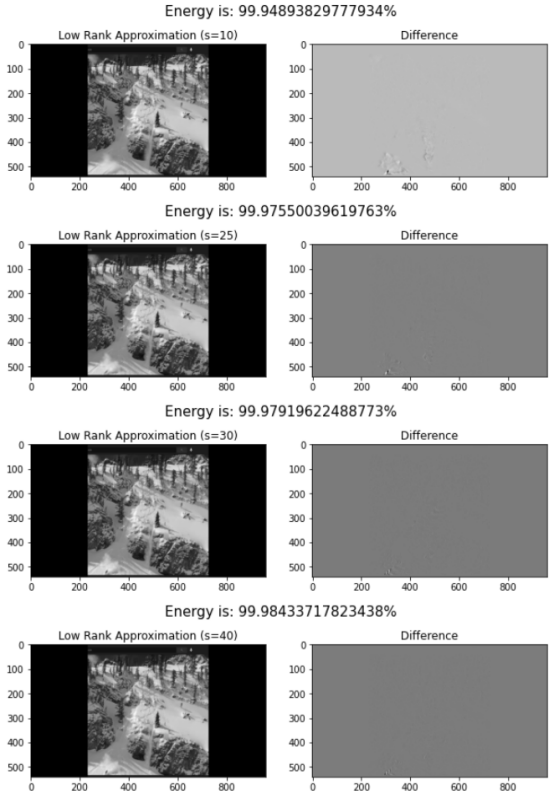


Figure 10: Result of the blocked randomized method with $b = 4$. Case: $s = 30$, $p = 10$, and $q = 1$.



(a)

Figure 9: In this figure 425^{th} frame of the Ski Drop video is used (for the main reason that in this frame we can see the man at the bottom) to test the low-rank approximation using different energy levels. The right column shows the difference between the original frame and low-rank approximation.